Unit Conversions

someonesdad1@gmail.com 23 Jan 2014

This document discusses the `u.py` module, which is a python library to provide numerical conversion factors for various measurement units. It's part of the `u.zip` package from http://code.google.com/p/hobbyutil/.

Installation is simple: put the `u.py` file in a directory in your `PYTHONPATH`. Then use `import u` when you want to use its facilities.

This library is derived from the ideas and code in the 1.13 version of `numericalunits.py` script of Steve Byrnes, located at PyPi (https://pypi.python.org/pypi/numericalunits).

Module overview

Features

1. Is easy to use (you can learn to use it in a few minutes by seeing an example or two).
2. Helps make code self-documenting.
3. Provides conversion factors to convert amongst commonly-used SI and non-SI units.
4. Easily change the set of units used or define new units.
5. All legal SI prefixes are usable with any unit[1].
6. Can determine when a dimensional error has been made in a calculation.
7. It's thread-safe.
8. Should work with any python version (tested with 2.6.5, 2.7.2, and 3.2.2).
9. The module is in python, but the same calculation technique will work with virtually any programming language.
10. The python code can call any other numerical calculation routines in another process, language, or computer and can still detect when dimensional errors have been made. This is because the dimensions are encoded in the floating point numbers when the randomization feature is used (see the ***Dimensional checking*** section).
11. If you're willing to download the python uncertainties library, you can perform numerical calculations using both dimensional units and uncertainties in the numerical values. Until we have this functionality built into a programming language, this is a useful substitute.

The default set of units includes a number of units along with common aliases. For example, the string `L` (abbreviation for liter) can also be given as `liter`, `liters`, `litre`, and `litres`. This lets calling code (i.e., various users) use the strings they are comfortable with to identify the desired units (which makes it less likely your code will break because users used something slightly different than what you anticipated). There's little overhead associated with defining aliases, so you should feel free to use as many as you wish (on my older computer, the processing time overhead to read in 1000 unit definitions is on the order of 15 ms).

I've made the default behavior to have angles and solid angles be measured in base units of `radians` and `steradians`. In addition, a `nodim` base unit is defined to let you track things that are dimensionless by turning them into something with dimensions. If you don't want such behavior, it's easy to change the definitions in the `SI_base_units` and `SI_unit_definitions_string` global variables.

Why you might want to use it

The `u.py` module lets you look up conversion factors for units for use in your python code. Once you see an example of use, you'll just start using the module with no further need of documentation.

1 With one exception: you can't use prefixes with the kg unit because this violates standard SI usage.

It's easy to see what's going on in your code.

Here's an example that shows defining a speed in units of feet per second and printing the output value in miles per hour (the module's methods are shown in color):

```
from u import u, to
speed = 88*u("feet/second")  # Speed will subsequently be in SI units
print("Speed in miles per hour is " + str(to(speed, "miles/hour")))
```

When you run this script, you'll get the output

```
Speed in miles per hour is 60.0
```

The majority of the time you won't need to bother with the dimensional checking feature -- you'll just write your code to make the calculations needed -- and what you just saw as an example is essentially all you need[2]. However, when a complicated function is called or you're using some code written by someone else, you can check that things are dimensionally-consistent by using the checking feature. It's not as convenient to use as other unit libraries, but it's useful to verify some code works correctly with respect to use of dimensions. To learn more about the dimensional checking feature, see the ***Dimensional checking*** section below.

A benefit of using a module that lets you convert amongst units easily is that you can use equations as you find them in references (as long as you know what units they are valid with). This is often a source of error when one has a bunch of program variables in one set of units and the formula needs another (with e.g. empirical formulas especially common in various engineering disciplines). However, using the unit conversion features of the units program, you make the needed conversions when the formula is evaluated. Manually converting the formula to another set of units often results in a mistake which can be hard to find -- or, you're just too trusting of your manual algebra and arithmetic skills and repeatedly make the same error.

It is easy to define base units and derived units and use `u.py` for subsequent calculations. You can read the docstrings and see the notes in the section ***Defining your own units*** about how to do such things. For most day-to-day use, the built-in units are probably adequate.

What's missing

Currently, I see the following disadvantages of the `u.py` module:

1. Calculations with numbers with units don't have different types (a feature of more complicated unit modules); this can make it easier for the code to spot unit problems.
2. A dimensional error isn't flagged as to the line that caused the problem.
3. The set of units aren't tested for consistency as some unit programs do.
4. Arbitrary expressions of units cannot be handled.

The last disadvantage is demonstrated by trying to use the unit of `(m/s)**2`; this will result in an exception at runtime. You can use `m**2/s**2`, so you'll have to manually translate such expressions. I may fix this last disadvantage in a future release (if you need this feature, send me an email).

I don't have enough experience with `u.py` yet to see if I'll stick with it as my unit library of choice for python code. I've already used it in a few applications and was pleased with the ease of use and the increased facility for getting correct numerical results. I haven't used the randomization technique yet to verify dimensional correctness as I prefer to use code inspections and test cases that I work out manually. But I expect there will come a time where things are a bit too complicated and I'll need the dimensional checking ability. I'll update these opinions after a year or so of use.

Other python modules dealing with units

If you need more sophisticated dimensional behavior, there are a number of python unit libraries to

---

2 Another function you might want to use if your program gets input from a user is the `ParseUnit()` function.

consider (these aren't recommendations, they're just relevant things I found in a search; and I haven't tried any of them):

- `Magnitude` (http://juanreyero.com/open/magnitude)
- `numericalunits` (http://pypi.python.org/pypi/numericalunits)
- `Pint` (https://pypi.python.org/pypi/Pint/)
- `pyuom` (https://github.com/katerina7479/python-units-of-measure)
- `Quantities` (https://pypi.python.org/pypi/quantities)
- `si` (http://christian.amsuess.com/tools/si/)
- `units` (http://www.nongnu.org/pyformex/doc/ref/units.html)
- `Unum` (http://home.scarlet.be/be052320/Unum.html)

For an article that discusses `numericalunits`, `Pint`, and `Units`, see http://www.drdobbs.com/jvm/quantities-and-units-in-python/240161101?pgno=1.

Examples

Listing the built-in units

Run the `Dump()` function and it will return a string listing of the supported units and their numerical conversion factors to the SI base units. Pass in a stream and this list will be printed to that stream.

Day-to-day use

The example given in the ***Why you might want to use it*** section above demonstrates most of what you need to know to use the module.

The majority of the time that I use the `u.py` module, I'm getting unit conversion factors for the programs I write. A handy use is to allow a program's users to enter numbers with physical dimensions in the units they are interested in. Without the facilities of the `u.py` module, the programmer has to write specialty code for each program (which, of course, demonstrates the desirability of having a library). I've tried to add all the units I'm likely to use in the general applications I use. You specialized scientists/engineers will of course want to add your units and constants (for example, many atomic/nuclear constants and units are missing). Just append your new unit definitions to the `SI_unit_definitions_string` global variable and run `Initialize` again.

A common task is to write an application that prompts a user for an input string. For example, an ideal gas law program might prompt the user for a pressure value. It is convenient to let the user also enter units of his choice in the response. The program would then pick apart the user's response. For example, the user might have typed in `3.7e3 Pa`. It can be a fair bit of work to accommodate a variety of units, even in just one program. But it's easy for a library like `u.py`. You can use the `ParseUnit()` function in `u.py` to pick apart the user's response. It will return the tuple `("3.7e3", "Pa")` whether the user typed in "`3.7e3Pa`", "`3.7e3 Pa`", or "`3.7e3  Pa`". Note it uses a keyword argument `allow_expr` to allow python expressions to be used with units; the expression and unit strings must be separated by one or more space characters.

nodim

The default set of units in the `u.py` script comes with a base unit called `nodim`. This is a bit of a misnomer, as it is in fact a base dimension on the same footing as the SI base units. The reason for doing this is to allow you to define and track calculations with numbers that are dimensionless. Of course, we're enabling this by giving the thing a dimension!

A justification for doing this is that you can use this feature to perform dimensional checking on your calculations and determine if the dimensionless variables have been accidentally combined with one

or more other base units, as demonstrated in the ***Dimensional checking*** section below.

There's little overhead for adding more base units, so use them if you see the need (a calculation example doing this is given in the ***Adding base units*** section.

Defining your own units

The `u.py` module lets you define your own unit system. This is done in an example in the unit tests, which defines a unit length in terms of the height of a pear tree and mass in terms of a cat. While perhaps a bit whimsical, it does illustrate that special units can be defined and used. Personally, I rarely use this feature because I am so accustomed to using SI and other non-SI units. However, you occasionally need such a feature and it was easy to add because of the module's design.

Dimensional checking

Steve Byrnes' clever idea was to use a random number generator to create "orthogonal" values for the base units of a system of units. When a variable is then multiplied by some combination of base units, **this unit combination is encoded in the floating point number**. Byrnes pointed out in his documentation that his tool isn't appropriate for beginners studying the use of units (say, high school or beginning college students), but it would be appreciated by working scientists and engineers. However, the underlying idea is pretty simple -- so if beginners can understand the following material, they might find this method a useful tool for their work.

In this section, we'll look at an example and an algebraic justification of the principle.

The following is a minimal example in python that illustrates the principles. You can step through the code with a debugger and see how things work. Start with a python library module called `un.py`:

```
from random import uniform

def Initialize():      # Define fundamental units
    global m, kg       # Put these symbols in the global namespace
    m  = 10**uniform(-1, 1)  # Random number between 0.1 and 10 inclusive
    kg = 10**uniform(-1, 1)
    MakeDerivedUnits()

def MakeDerivedUnits():
    global mm
    mm = 1e-3*m
```

We'll utilize this module in the `demo.py` script:

```
import un

# You must call un.Initialize () before performing unit calculations
un.Initialize()

# Suppose we have a square with a side of 1.7 m.  Let's use the un
# module to help us calculate the area in square mm.
s_m = 1.7*un.m
area_sq_m = s_m*s_m
print("Area in square mm = " + str(area_sq_m/un.mm**2))

# To demonstrate dimensional checking, display the area in square kg.
print("Area in square kg = " + str(area_sq_m/un.kg**2))
```

If you run this script, you'll see something like

```
Area in square mm = 2890000.0
Area in square kg = 779.842032705
```

although it's unlikely you'll get the same second number as I did. The **key idea** is to run this script again:

```
Area in square mm = 2890000.0
Area in square kg = 0.00631150267997
```

Note the highlighted numbers for the area in square kg differ substantially, but the area in square mm is the same for both calculations. **This difference in successive runs indicates a dimensional error was made in the calculation** using square kg -- it indicates that the dimensions are not consistent.

Because the `u.py` module is a lightweight module, it cannot tell you where the error was made. More sophisticated unit libraries can do such things. However, when you're writing code doing numerical calculations with units, you'll be aware of the units you've used for variables and you'll likely encode those units in the variable name to help with documentation. For example, the above code shows the area is calculated in square meters: `area_sq_m`. Then dimensionally checking your code while developing it should let you find where any problems are.

An implementation detail between the above example and the `u.py` script is that I chose not to use global variables for the conversion factors in the `u.py` script like the `numericalunits` module does. The main reason was to make things a little less clumsy when using SI prefixes, which I use a lot. However, if I didn't need a large numerical range of values, I'd switch the `u.py` script's architecture to use global variables as was done in the `numericalunits` module, as they are notationally a little more convenient and let you do away with using the module's `u()` function.

The algebra

Let's show how this dimensional checking works using algebra. Define the three conversion factors

μ_m = conversion factor to meters $\neq 0$
μ_{mm} = conversion factor to millimeters $= \alpha\,\mu_m \neq 0$ and $\alpha \neq 0$
μ_{kg} = conversion factor to kilograms $\neq 0$
Note $\mu_m \neq \mu_{mm} \neq \mu_{kg}$

Let's repeat the calculation done above in the code example. Suppose we define a length s_m of the side of the square in meters:

$$s_m = \beta\,\mu_m$$

β is a number and in the above python code, it was 1.7. The area A_{sq_m} of the square in square meters is

$$A_{\text{sq\_m}} = s^2 = \beta^2\,\mu_m^2$$

To convert A_{sq_m} to square millimeters, we divide by μ_{mm}^2

$$A_{\text{sq\_mm}} = \frac{A_{\text{sq\_m}}}{\mu_{mm}^2} = \frac{s^2}{\mu_{mm}^2} = \frac{\beta^2\mu_m^2}{\mu_{mm}^2} = \frac{\beta^2\mu_m^2}{(\alpha\,\mu_m)^2} = \left(\frac{\beta}{\alpha}\right)^2$$

To convert A_{sq_m} to square kilograms, we divide by μ_{kg}^2 (note this is a dimensionally-inconsistent thing to do)

$$A_{\text{sq\_kg}} = \frac{A_{\text{sq\_m}}}{\mu_{kg}^2} = s^2 = \frac{\beta^2\mu_m^2}{\mu_{kg}^2} = \beta^2\left(\frac{\mu_m}{\mu_{kg}}\right)^2$$

The last terms in color are unequal. Also note that the expression for $A_{\text{sq\_mm}}$ is independent of any of the μ's, so it has the same value regardless of the numerical values of the base units (this is a characteristic of being dimensionally consistent). However, the expression for $A_{\text{sq\_kg}}$ depends on the μ's. Thus, when the μ's are set to different random numbers on subsequent runs, **the non-dimensionally-consistent values will differ**, flagging the dimensional inconsistency. It's simple and elegant.

All of the length units defined with respect to meters contain the factor μ_m. This lets you recover the correct numerical value in the units you want, as long as you remember to divide by the factor for the desired units, even when the μ's are defined to be random numbers -- but only when the calculations are dimensionally correct, as shown by the algebra.

A caveat is that the method depends on the μ's getting non-equal values from the random number generator. Hence, the correct statement is that if you get different values when using the random number generator, you're *almost* certain that a dimensional error has been made. This statement applies to the `numericalunits.py` module, but the `u.py` module has code that ensures the random numbers used for the base units are unique. This extra code improves the "almost certain", but still can't guarantee certainty. If you're uncertain and want to be really sure, run the calculation a number of times (you'll have to define "a number of", but 3 to 5 times might be a good choice) and make the comparisons. Because python's random number generator is a good one that's well-studied, it would be extraordinarily surprising to see the script not be able to detect a dimensional error.

A practical issue: roundoff error

Dimensional-checking calculations with the `u.py` script can indicate a dimensional inconsistency when in fact one doesn't exist. This situation is caused by roundoff error. An example is to set `self.eps` to zero in the script's unit test section and run the script. You'll see a number of test cases fail. The cause is the non-associativity of floating point calculations. The fix is straightforward: define a difference only when the two calculations differ by a large enough value (the `assertAlmostEqual()` method overrides the base class definition to provide this facility).

If you're interested in more details, see "Accuracy of Floating Point Arithmetic", section 4.2.2 of Knuth's *Seminumerical Algorithms*. Also see the section "Roundoff error" in `DinosaurArithmetic.pdf` at http://code.google.com/p/hobbyutil/.

Adding base units

It is easy to add additional base units to your defined units. I'll illustrate this with an example involving feeding cats and dogs. It demonstrates how adding new base units to the built-in SI units is done and how it can help catch errors in computer calculations.

The problem is simple: given a number of dogs and cats and the mass of food required to feed each animal, calculate the total mass of food required to feed the animals. The problem is simple to calculate in your head, but I will illustrate it with a script that contains an intentional error. We'll use `u.py` to help discover the error.

Each dog gets 0.2 kg of food and cats get 0.1 kg of the same food. We have to feed 7 dogs and 12 cats. How much food do we need? Clearly, the answer is

$$(7\ \text{dogs})(0.2\ \text{kg/dog})+(12\ \text{cats})(0.1\ \text{kg/cat}) = 2.6\ \text{kg}$$

Note the use of "dogs" and "cats" as "units" to aid the calculation. This is a powerful and helpful way to do many calculations, as the following example illustrates.

The mistake we'll make is to swap the numbers for the animals and instead calculate

$$(12\ \text{cats})(0.2\ \text{kg/dog})+(7\ \text{dogs})(0.1\ \text{kg/cat}) = 3.1\ \text{kg}$$

The total food used would then be 20% higher than it should be. Note how putting the units in makes the error obvious, but we'll assume we didn't see it and continue with the calculation. It's a bit unnerving to realize that the vast majority of computer programs proceed in calculations involving numbers with units with absolutely no safety net.

Here's a script that demonstrates this calculation, along with some dimensional consistency checking:

```
import sys, u

# Make dogs and cats independent base units
u.base_units.add("cat")
u.base_units.add("dog")
# Re-initialize to incorporate these new units
u.Initialize()

# Do the correct calculations
```

```
food_per_dog = 0.2*u.u("kg/dog")
food_per_cat = 0.1*u.u("kg/cat")
# Number of animals
n_dogs =  7*u.u("dog")
n_cats = 12*u.u("cat")
# Total food amount needed in kg
total_kg = n_dogs*food_per_dog + n_cats*food_per_cat
# Report results
print("Total (correct value) food in kg = %.3g" % total_kg)

def BadCalculation(randomize=False):
    # Make the deliberate mistake of swapping the cat and dog
    # numbers.
    u.Initialize(randomize=randomize)
    # Set up our variables (THIS IS THE ERROR)
    n_dogs = 12*u.u("cat")
    n_cats =  7*u.u("dog")
    # Total food amount needed in kg
    total_kg = n_dogs*food_per_dog + n_cats*food_per_cat
    # Report results
    print("Total (mistake; randomize is %s) food in kg = %.3g" %
        (randomize, total_kg))

BadCalculation(randomize=False)
BadCalculation(randomize=True)
```

The code in error is shown in red. I ran this script and got the following results:

```
Total (correct value) food in kg = 2.6
Total (mistake; randomize is False) food in kg = 3.1
Total (mistake; randomize is True) food in kg = 1.88
```

Running it again gave the results:

```
Total (correct value) food in kg = 2.6
Total (mistake; randomize is False) food in kg = 3.1
Total (mistake; randomize is True) food in kg = 0.959
```

The numbers are as we predicted. However, the important numbers are in red, showing that two supposedly identical runs of the same code gave different answers (if you run the same code, you will get different numbers than I did). **This indicates we made a dimensional mistake**.

> A key lesson to draw from this example is that you can add independent base units to the `u.py`'s units and use them to help verify your program's calculations are correct. These base units can be **anything** you consider "orthogonal" to other things, such as buildings, number of welders, number of machinists, loaves of bread, etc. Another benefit of using the `u.py` library is that it helps make your code more readable, which means it's easier to maintain and visually inspect for correctness.

Don't hesitate to add more base units -- the overhead is negligible compared to the importance of finding mistakes.

Using with the uncertainties library

An important addition to python's capabilities is the python `uncertainties` library by Eric Leibigot (see http://pythonhosted.org/uncertainties/). This library assists you in doing linear uncertainty propagation; it's easy to use and has powerful features.

Uncertainty propagation seems to be about as poorly-learned today as it was when I was a student. Even worse, many technical people don't understand the reasons behind you'd want to use it:

1. You want to make the best decisions you can from experimental data.
2. Help with experimental design; improve an existing experiment.
3. Let others use your data and "stand on your shoulders".

Frankly, the majority of scientific and engineering papers I've seen do a middling to poor job with

respect to uncertainty -- and it's not helped by the editors and referees, as they may not know any better themselves. If you're a technical person, you'd be wise to take an undergraduate engineering statistics course, as it will teach you most of what you need to know and you'll use the material for the rest of your career. After you've finished the course, take a look at the GUM (http://www.bipm.org/en/publications/guides/gum.html) and most of it should make sense. Then you can confidently use the `uncertainties` library.

You can look at the `u.py` module's test code (the `testWithUncertainties` function) and see an example of using units with uncertainties. In fact, the unit is encoded in the nominal value of the `ufloat` (a model of a random variable) and calculations proceed just like they do with regular floating point numbers.

Here's another example. The http://physics.nist.gov/constants website gives the following values for some physical constants (current as of 23 Jan 2014):

Avogadro's number	$N_A = 6.02214129(27)\times10^{23}$
Boltzmann constant	$k = 1.3806488(13)\times10^{-23}$
Molar gas constant	$R = 8.3144621(75)$

The uncertainty of each value is shown in parentheses and represents one standard deviation in the two least significant figures in the significand. Thus, the uncertainty for Avogadro's number is 0.00000027×10^{23}. This is a common short-hand notation for expressing an experimentally-determined value and its uncertainty.

The Boltzmann constant is the molar gas constant per molecule, so it can be calculated by the expression R/N_A. We'll use the following script to calculate k by this expression and see that its uncertainty is essentially the same as what the NIST value is. In addition, we'll encode the nominal values of the `ufloat`s with the appropriate units and use them to print out the value of k in eV/K (see http://physics.nist.gov/cuu/Constants/Table/allascii.txt for an extensive listing of constants in ASCII form; the expected eV/K value came from this table). You could also perform dimensional consistency checks on these random variates if you used `Initialize(randomize=True)`, but we won't do that here.

```
from uncertainties import ufloat, ufloat_fromstr
from u import u, to

NA = ufloat_fromstr("6.02214129(27)e23")
k  = ufloat_fromstr("1.3806488(13)e-23")
R  = ufloat_fromstr("8.3144621(75)")

# Attach units to the nominal values
NA *= u("1/mol")
k  *= u("J/K")
R  *= u("J/(mol*K)")

print("Avogadro's constant  = " + str(NA))
print("Molar gas constant   = " + str(R))
print("Boltzmann's constant (NIST website) = " + str(k))
print("Boltzmann's constant (calculated)   = " + str(R/NA))
k_eV_per_K = to(k, "eV/K")
print("Boltzmann's constant in eV/K (calculated) = " + str(k_eV_per_K))
expected = ufloat_fromstr("8.6173324(78)e-5")
print("Boltzmann's constant in eV/K (expected)   = " + str(expected))
```

When this script is run, the results printed are

```
Avogadro's constant  = (6.02214129+/-0.00000027)e+23
Molar gas constant   = 8.314462+/-0.000008
Boltzmann's constant (NIST website) = (1.3806488+/-0.0000013)e-23
Boltzmann's constant (calculated)   = (1.3806488+/-0.0000012)e-23
Boltzmann's constant in eV/K (calculated) = (8.617332+/-0.000008)e-05
Boltzmann's constant in eV/K (expected)   = (8.617332+/-0.000008)e-05
```

I'm not sure of the source of the highlighted discrepancy, but it would be unimportant in most practical situations.

Note you can attach units to the random variables either in the `ufloat()` call itself for the nominal value (first parameter) or after the random variable is defined, as shown in the above example.